A Simple 'Linearized' Learning Algorithm which Outperforms Back-Propagation[1]

J.R. Deller, Jr. and S.D. Hunt

Michigan State University
Department of Electrical Engineering / 260 EB
and Signal Processing Group: Speech Processing Laboratory
East Lansing, MI 48824-1226 USA
author_surname@ee.msu.edu

Abstract: A class of algorithms is presented for training multilayer perceptrons using purely "linear" techniques. The methods are based upon linearizations of the network using error surface analysis, followed by a contemporary least squares estimation procedure. Specific algorithms are presented to estimate weights node-wise, layer-wise, and for estimating the entire set of network weights simultaneously. In several experimental studies, the node-wise method is superior to back-propagation and an alternative linearization method due to Azimi-Sadjadi *et al.* in terms of number of convergences and convergence rate. The layer and network-wise updating offer further improvement.

1. Introduction

This paper introduces a new class of learning algorithms for feedforward neural networks (FNN) with improved convergence properties. In spite of the nonlinearities present in the dynamics of a FNN, the learning algorithm is purely "linear" in the sense that it is based on a contemporary version (see [1]) of the recursive least squares (RLS) algorithm (e.g. [2]). Accordingly, unlike the popular back-propagation algorithm used to train FNNs [3, 4], the new learning algorithm and its potential variants will benefit from the well-understood theoretical properties of RLS and VLSI architectures for its implementation.

A FNN is an artificial neural network consisting of nodes grouped into layers. In this paper, we consider a two-layer network[2], but the generalization of the method to an arbitrary number of layers is not difficult. Working from the bottom up, we shall frequently refer to layers zero, one, and two as the "input," "hidden," and "output" layers, respectively. Each node above the input layer in the FNN passes the sum of its weighted inputs through a non-linearity to produce its output. The inputs to the input layer are the external inputs to the network, and the outputs of the output layer are the external outputs.

The number of nodes in layer i is denoted N_i, with N_0 indicating the number of input nodes at the bottom of the network. The weight connecting node j in the hidden layer to node k in the output layer is denoted $w_{k,j}$. The weight connecting input node l to node j in the hidden layer is denoted $w'_{j,l}$. We denote by N the number of *training patterns* of the form

$$\{(x_1(n), x_2(n), \ldots, x_{N_0}(n); t_1(n), t_2(n), \ldots, t_{N_2}(n)), \quad n = 1, 2, \ldots, N\}, \tag{1}$$

in which $x_l(n)$ is the input to the l^{th} node in layer zero, and $t_k(n)$ is the *target* output for node k in the output layer (output *desired* in response to the corresponding input). The *computed* outputs of layer two [one] in response to $x_1(n), \ldots, x_{N_0}(n)$ are denoted $y_1(n), \ldots, y_{N_2}$ $[y'_1(n), \ldots, y'_{N_2}]$. Finally, we need to formalize the nonlinearity associated with the nodes. Consider node k in the output layer. For given weights, $w_{k,j}$, $j \in [1, N_1]$, the output in response to the n^{th} input is

$$y_k(n) = S\left(\sum_{j=1}^{N_1} w_{k,j} y'_j(n)\right) \tag{2}$$

in which $S(\cdot)$ is a differentiable nonlinear mapping. For future purposes, we define $\dot{S}(\cdot)$ to be the derivative of $S(\cdot)$. For convenience, we also define $u_k(n) \stackrel{\text{def}}{=} \sum_{j=1}^{N_1} w_{k,j} y'_j(n)$. Clearly, $u_k(n)$ is the input to node k in the output layer in response to pattern n. $u'_l(n)$ is similarly defined as the input to node l in the hidden layer.

[1] **Acknowledgements:** This work was supported by the Office of Naval Research under Grant No. N00014-91-J-1329, and by the National Science Foundation under Grant No. MIP-9016734. JD was also supported by an Ameritech Fellowship and SH by a fellowship from the University of Puerto Rico.

[2] Some authors might choose to call this a *three* layer network. We shall designate the bottom layer of "nodes" as "layer zero" and not count it in the total number of layers. Layer zero is a set of linear nodes which simply pass the inputs unaltered.

Many training (weight estimation) algorithms exist for this type of network (e.g. [3] – [7]). The most popular, the back-propagation algorithm [3], [4], performs satisfactorily in some cases if given enough time to converge. However, the literature abounds with example applications in which back-propagation convergence is too slow for practical usage (e.g. see [8]). One attempt to develop faster training methods is represented by the class of algorithms in which the network mapping is "linearized" in some sense in order to take advantage of linear estimation algorithms. It is with this class of algorithms that this paper is concerned.

2. Linearization Algorithm

The fundamental training problem for the two layer FNN is stated as follows: Given a set of N training patterns as in (1), find the network weights which minimize the sum of squared errors, $E = \sum_{n=1}^{N} \sum_{k=1}^{N_2} \lambda(n)(t_k(n) - y_k(n))^2$,where the weights $\lambda(\cdot)$ are included for generality. For a given set of training pairs, E is a function of the weights of the network. A graph of E over the weight space is frequently called an *error surface*. Ideally, a training algorithm would find the weights corresponding to the global minimum of the error surface. Training algorithms usually operate by sequentially presenting the training patterns and moving the weights toward a minimum of the error surface. The procedure is repeated several times using different initial weights in order to locate the best minimum. Ideally, *all* weights will be altered with each presentation of the set of training patterns so that the weights may move in the direction of steepest descent. In this case the algorithm represents a true *gradient descent* approach. In practice, however, no reasonable algorithm exists which can simultaneous change each weight in the network. In fact, the popular back-propagation algorithm works on only one weight at a time. One of the principal benefits of the method to be presented here is that many weights can be simultaneously updated.

The linearization technique adopted in this work can be explained in terms of error surface analysis. In effect, for a present set of weights and a given training pattern, we construct a "linearized" network with an error surface, say $\bar{E}$, which is "similar" in some sense to E in a neighborhood of the present weights. There are two similarity criteria: first, that the magnitude of E and $\bar{E}$ be the same at the present weights; and second, that the derivatives of E and $\bar{E}$ with respect to the weights *to be updated* be the same at the present weights (since the other weights are not altered, it is not necessary that the derivatives with respect to those weights match).

Let us digress momentarily from the simple two layer network and use more general description. Suppose that the weights connected to one or more nodes in layer L are to be updated simultaneously[3]. This may include as few as one, and as many as all, nodes in layer L. Denote the set of such selected nodes by $\mathcal{N}$. Denote by $\mathcal{M}$ the set of all nodes above layer L to which any node in $\mathcal{N}$ is connected, directly or indirectly. Let all weights not connected to nodes in $\mathcal{N}$ and $\mathcal{M}$ be fixed at present values[4]. Then it is shown in [9] that a "linearized" network whose error surface $\bar{E}$ is similar to E in the senses above is constructed by replacing the nonlinearity $S(\cdot)$ for each node in $\mathcal{N}$ and $\mathcal{M}$ by a linear approximation, say $\hat{S}(\cdot)$, consisting of the first two terms of a Taylor series around the "present" value of the node's input. For example, suppose the k^{th} output node is to be linearized with respect to the n^{th} training pattern. Let $\tilde{w}_{k,j}$ denote the present value of weight $w_{k,j}$. Then,

$$\begin{aligned} S(u) \;\approx\; \hat{S}(u) &= \dot{S}\left(\sum_{j=1}^{N_1} \tilde{w}_{k,j} y_j'(n)\right)\left[u - \sum_{j=1}^{N_1} \tilde{w}_{k,j} y_j'(n)\right] + S\left(\sum_{j=1}^{N_1} \tilde{w}_{k,j} y_j'(n)\right) \qquad (3) \\ &= \dot{S}\left(\sum_{j=1}^{N_1} \tilde{w}_{k,j} y_j'(n)\right) u + \left[S\left(\sum_{j=1}^{N_1} \tilde{w}_{k,j} y_j'(n)\right) - \dot{S}\left(\sum_{j=1}^{N_1} \tilde{w}_{k,j} y_j'(n)\right) \sum_{j=1}^{N_1} \tilde{w}_{k,j} y_j'(n)\right] \stackrel{\text{def}}{=} K_k(n)u + b_k(n). \end{aligned}$$

In fact, since $S(u) = \hat{S}(u)$ if u is the input corresponding to the present weights, any node not in $\mathcal{N}$ or $\mathcal{M}$ may also be linearized with no effect on the solution. Therefore, we may assume without loss of generality that the entire network is linearized, even if only a portion of the weights is to be updated.

It will become clear below that once the network is linearized by replacing the operation $S(\cdot)$ by $\hat{S}(\cdot)$ in all appropriate nodes, in principle any least square error algorithm can be used to update the weights. Algorithms based on similar ideas for updating weights one node at a time are given by Azimi-Sadjadi *et al.* [5] (henceforth, *A-S algorithm*) and by Hunt and Deller [9]. The former is based on the conventional RLS algorithm [2] with a

[3] If any weight connected a node is to be updated, then every weight connected to that node must be updated. This "constraint" is ordinarily beneficial, since it implies the ability to simultaneously update more than one weight.

[4] In certain cases it is possible to update weights in different layers simultaneously. We discuss one case at the end of this section.

forgetting factor, while the latter employs a contemporary QR decomposition algorithm [1, 10] for significant performance improvement. The view of the method taken above allows us to to further exploit the linearization by complete *layer*-wise updating of weights for even further improvement. Let us pursue this layer-wise approach.

Suppose we wish to update all weights in the output layer simultaneously. We must linearize all output nodes (and may arbitrarily linearize any other nodes). For node k in the output layer, the output in response to input n is computed as in (2). Let $\bar{y}_k(n)$ represent the output of node k after $S(u_k(n))$ as been replaced by $\hat{S}(u_k(n)) = K_k u_k(n) + b_k$. Accordingly,

$$\bar{y}_k(n) = K_k(n)[\sum_{j=1}^{N_1} w_{k,j} y'_j(n)] + b_k(n) \quad \text{or} \quad \bar{z}_k(n) = K_k(n)[\sum_{j=1}^{N_1} w_{k,j} y'_j(n)] \tag{4}$$

with $\bar{z}_k(n) \stackrel{\text{def}}{=} \bar{y}_k(n) - b_k(n)$. We speak of the rightmost form in (4) as descriptive of a *linearized node* since the output is a purely linear combination of the inputs to the node. The network with all appropriate nodes linearized will be called the *linearized network*. Since $\bar{y}_k(n) = y_k(n)$ at the present weights, the error at the k^{th} node will be the same for the linearized and original network if the target value for $\bar{z}_k(n)$, say $\bar{t}_k(n)$, is taken to be

$$\bar{t}_k(n) \stackrel{\text{def}}{=} t_k(n) - b_k(n) \tag{5}$$

and the linearized inputs to node k at pattern n are

$$\bar{x}_{k,j}(n) \stackrel{\text{def}}{=} K_k(n) y'_j(n), \quad j = 1, 2, \ldots, N_1. \tag{6}$$

Note that the linearized inputs are dependent upon k, so that we have effectively increased the number of training pairs by a factor of N_2.

The problem has effectively been reduced to one of estimating weights for a single-layer linear network. In order to simultaneously update the all weights in the output layer, the system of $N \times N_2$ equations

$$\bar{t}_k(n) = \sum_{j=1}^{N_1} \bar{x}_{k,j} w_{k,j}, \quad k = 1, 2, \ldots, N_2 \quad n = 1, 2, \ldots, N \tag{7}$$

must be solved for the least square estimate of the $N_1 \times N_2$ weights $w_{k,j}$, $k \in [1, N_2]$ $j \in [1, N_1]$. However, since all weights in the hidden layer are fixed, the outputs $y'_j(n)$ are independent of k. This means that the equations indexed by different values of k are independent of one another, and the sets of weights connected to different outputs may be updated independently. In the output layer, therefore, there is no theoretical difference between layer-wise and node-wise updating. This is not true at lower layers, however, as we now show for the hidden layer of the present network.

To update all weights in the hidden layer simultaneously, the weights in the output layer are fixed and *all* nodes in the network must be linearized. The outputs of the hidden layer with $S(\cdot)$ replaced by $\hat{S}(\cdot)$ are given by

$$\bar{y}'_j(n) = K'_j(n)[\sum_{l=1}^{N_0} w'_{j,l} x_l(n)] + b'_j(n), \quad j = 1, 2, \ldots, N_1. \tag{8}$$

Substituting (8) in the leftmost expression in (4) results in

$$\bar{y}_k(n) - [\sum_{j=1}^{N_1} K_k(n) w_{k,j} b'_j(n) + b_k(n)] = \sum_{j=1}^{N_1} \sum_{l=1}^{N_0} [K_k(n) w_{k,j} K'_j(n) x_l(n)] w'_{j,l}. \tag{9}$$

As above, we can now view the problem as one of training a single-layer linear mapping with target outputs

$$\bar{t}'_k(n) = t_k(n) - [\sum_{j=1}^{N_1} K_k(n) w_{k,j} b'_j(n) + b_k(n)] \tag{10}$$

and inputs

$$\bar{x}'_{k,j,l}(n) = K_k(n) w_{k,j} K'_j(n) x_l(n). \tag{11}$$

The weight estimates for $w'_{j,l}$, $j \in [1, N_1]$ $l \in [1, N_0]$ comprise the least square error solution to the system of equations

$$\bar{t}'_k(n) = \sum_{j=1}^{N_1} \sum_{l=1}^{N_0} \bar{x}'_{k,j,l}(n) w'_{j,l} \quad k = 1, 2, \ldots, N_2 \quad n = 1, 2, \ldots, N. \tag{12}$$

Unlike the output layer, we see that the problem cannot be decomposed into separate solutions for sets of weights connected to individual nodes in the hidden layer. This is a reflection of the fact that all weights in the hidden layer are coupled through their "mixing" in the output layer. This means that the simultaneous solution for all weights in the hidden layer should be beneficial with respect to a node-wise solution. Indeed we will find this to be the case in the experiments. Of course, this same intra-layer dependence of weights would continue if there were further hidden layers to be considered.

Note that, for a fixed k, the inputs to the linearized network, $\bar{x}'(n)$, $n \in [1, N]$, are most conveniently viewed as two-dimensional (indexed by couples (j, l)). There are N such "grid" inputs for each k, paired with the N values of $\bar{t}'_k(n)$. If there were further hidden layers in the network, we would find that the effective inputs would continue to increase in dimension. Further, it is noted that the role of k in (12) is somewhat superfluous. In principle, the index is used to keep track of which of N_2 outputs in the linearized network is being considered. However, the training pairs $(\bar{t}'_k(n); \bar{x}'_{k,1,1}(n), \ldots, \bar{x}'_{k,N_1,N_0}(n))$, $k \in [1, N_2]$ $n \in [1, N]$, can be reindexed by mapping pairs $(k, n) \rightarrow i$ so that the training pairs may be written $(\bar{t}'(i); \bar{x}'_{1,1}(i), \ldots, \bar{x}'_{N_1,N_0}(i))$, $i \in [1, N \times N_2]$. Of course, an identical system of equations to (12) results, but the linearized network may be viewed as a *single output* linear layer with $N \times N_2$ training pairs.

Updating of some subset of the weights in the hidden layer (in particular, "node-wise" as in the A-S algorithm) is tantamount to solving the subsystem of (12) corresponding to the desired weights, introducing the updated values into the system, solving for the next desired subset, etc. Clearly, this will result in a different solution than the simultaneous solution. In terms of the error surfaces, this process consists of continually updating the error surface as "partial" information becomes available, then moving in the direction of the gradient with respect to a new subset of weights in the updated surfaces. Intuitively, movement "at once" with respect to the "complete" gradient would seem to be a preferable procedure. Indeed, the later operation corresponds to the simultaneous updating.

The linearization allows us to approximate the error surface of the nonlinear system for only a small neighborhood around the present weights. Because of the criteria used to construct $\bar{E}$, the weights will be changed in the *direction* of the true gradient in the nonlinear space, but will move to the minimum of $\bar{E}$ which may be quite far from the neighborhood over which $E \approx \bar{E}$. Accordingly, the weights must be allowed to change only a small amount using the training patterns of the linearized system. If the linearized procedure results in a large change of weights, measures must be taken to decrease the alteration. The updating procedure is repeated until changing the weights does not result in a decrease in error. The algorithm proceeds as follows: linearize the system around the present weights, change the weights by a small amount to decrease error, then repeat the procedure. This is done until changing the weights does not decrease the error or a maximum on the number of linearizations is reached.

For the same reason that simultaneous *layer*-wise estimation of weights is beneficial, we should expect even more benefit from complete *network* updating if such were possible. It follows from the developments above that entire network updating is possible for at least one case. *If there is a single node in the output layer of the network*, let $k = 1$ and define

$$w^{\dagger}_{j,l} = w_{k,j} w'_{j,l} = w_{1,j} w'_{j,l} \tag{13}$$

From (9) it follows that

$$(\bar{y}_1(n) - b_1(n)) = \sum_{j=1}^{N_1} \sum_{l=1}^{N_0} [K_1(n) K'_j(n) x_l(n)] w^{\dagger}_{j,l} + \sum_{j=1}^{N_1} [K_1(n) b'_j(n)] w_{1,j}. \tag{14}$$

This can be interpreted as an attempt to train a single linear layer with one output and $(N_0 \times N_1) + N_1$ inputs. In this case, there will be only N linearized training patterns. The system can be solved for $w_{1,j}$ and $w^{\dagger}_{j,l}$, $j \in [1, N_1]$ $l \in [1, N_0]$ and (13) can be used to solve for $w'_{j,l}$, $j \in [1, N_1]$ $l \in [1, N_0]$.

3. Experimental Results

The results given in this section compare five training strategies for a FNN. These are: 1. Conventional back-propagation (no linearization in the sense described here, weight-wise updating); 2. A-S algorithm (node

Implementation	Back-Prop	A-S	Node Updating	Layer Updating	Network Updating
No. of Convergences	11	8	78	96	99

Table 1: Number of convergences per 100 sets of initial weights in experiments with the XOR network.

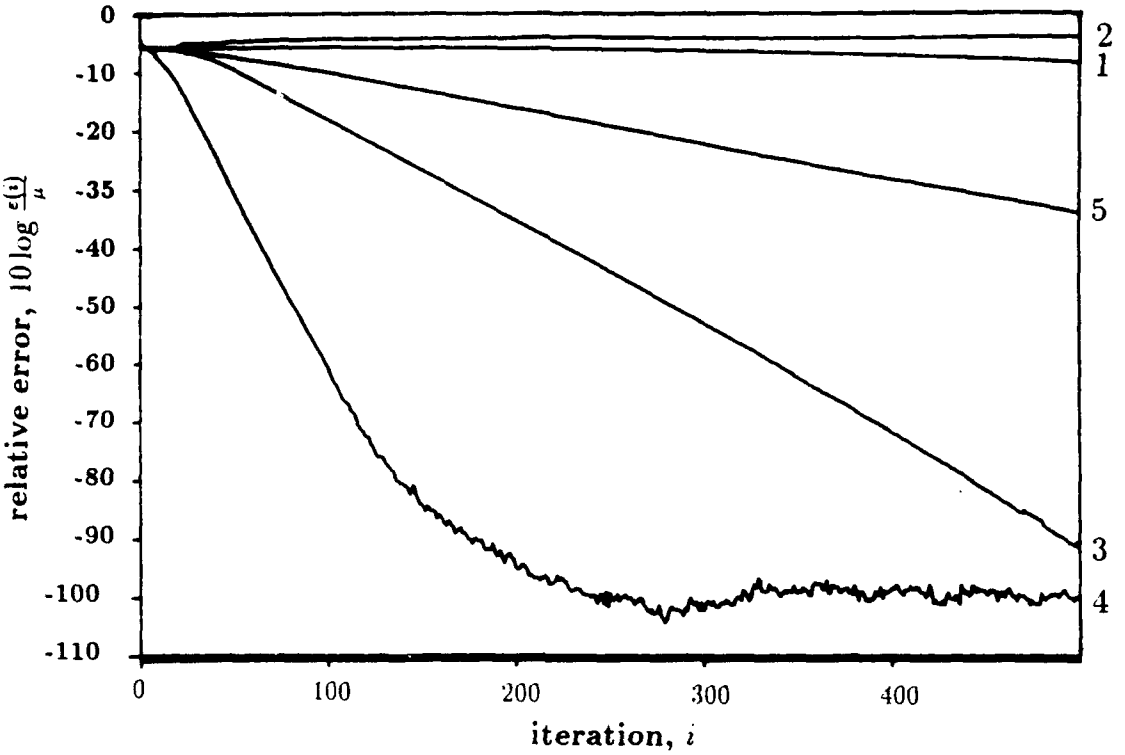

Figure 1: Average error in dB for the XOR implementations vs. iteration number. 1.Back-propagation; 2.A-S algorithm; 3. Node-wise updating; 4.Layer-wise updating; 5.Network wise updating.

linearization, then conventional RLS with a forgetting factor for node-wise updating); 3. Linearization method described above with node-wise updating based on QR decomposition; 4. Same as 3 with *layer*-wise updating; 5. Same as 3 with *complete network* updating. The two-bit parity checker (XOR) network used in the simulations has two inputs, two hidden layer nodes and one output node. An additional node is added at each layer whose output value was always unity, to serve as a bias for each node in the layer above. The initial weights were chosen as follows. Each weight in the network was selected randomly from a uniform distribution over the interval $[-1, 1]$. This procedure was repeated 100 times to select 100 sets of initial weights. The same 100 sets of weights were used for all five implementations. For the back-propagation algorithm, a factor of 0.04 was used in the weight updating equation. The A-S algorithm was implemented using no weight change constraints. The forgetting factor for A-S and for the QR decomposition implementation was 0.98. The QR decomposition implementation used a weight constraint of 0.2, meaning that the weight vector associated with each node was allowed to change at most by 0.2 in Euclidean norm during each iteration. The layer-wise updating algorithm has a forgetting factor of 0.3 and a weight constraint of 1.0. The network-wise updating algorithm had the same forgetting factor and weight constraint as the layer case.

Simulations were run to compare the number of times each implementation found weights that solve the XOR problem for the 100 initial weight sets. The results are shown in Table 1.

Simulations were also run to compare the output error of each algorithm. In the resulting figures, the error in dB means the following: Let $\varepsilon(i)$ be the sum of the squared errors incurred in iteration i through the training patterns, averaged over the 100 initial weight sets. Then, plotted in the figures is $10 \log(\varepsilon(i)/\mu)$ (dB), where μ is the maximum possible error in any iteration. Figure 1 shows the errors of the four XOR implementations.

4. Conclusions

A new implementation for node-wise weight updating algorithm for feedforward neural networks and new algorithms that update weights layer-wise and network-wise have been presented in this paper. The QR decomposition implementation has been shown experimentally to be superior to standard recursive equations for the node-wise updating algorithm. The layer-wise and network-wise weight updating algorithms were developed to improve the convergence rate and the speed of convergence. Both objectives were accomplished, with the layer-wise weight updating algorithm showing a significant advantage over both the single node weight updating algorithm used as a reference, and the widely used back-propagation algorithm.

References

[1] J.R. Deller, Jr. and D. Hsu, "An alternative adaptive sequential regression algorithm and its application to the recognition of cerebral palsy speech," *IEEE Trans. Circuits and Systems*, vol. CAS-34, pp.782-786, July 1987.

[2] D. Graupe, *Time Series Analysis, Identification and Adaptive Filtering*, Malabar, Florida: Krieger Publishing Company, 1989.

[3] P. Werbos, *Beyond Regression: New Tool for Prediction and Analyses in the Behavioral Sciences* (Ph.D. dissertation), Harvard University, Cambridge, Massachusetts, 1974.

[4] D. Rumelhart, G. Hinton, and R. Williams, "Learning internal representations by error propagation," in D. Rumelhart and J. McCleland (editors), *Parallel Distributed Processing*, vol. 1 Cambridge, Massachusetts: MIT Press, 1986.

[5] M. Azimi-Sadjadi, S. Citrin, and S. Sheedvash, "Supervised learning process of multi-layer perceptron neural networks using fast least squares," *Proc. IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, pp.1381-1384, 1990.

[6] S. Kollias and D. Anastassiou, "An adaptive least squares algorithm for the efficient training of artificial neural networks," *IEEE Trans. Circuits and Systems*, vol. CAS-36, pp.1092-1101, August 1989.

[7] M. Azimi-Sadjadi and S. Citrin, "Fast leaning process of multi- layer neural nets using recursive least squares technique," *Proc. IEEE Int. Conf. on Neural Networks*, Washington D.C., June 1989.

[8] R. Lippmann, "Review of neural networks for speech recognition," *Neural Computation*, vol. 1, pp. 1-38, 1989.

[9] S.D. Hunt and J.R. Deller, Jr., " 'Linearized' alternatives to back-propagation based on recursive QR decomposition," *IEEE Trans. Neural Networks* (in review).

[10] G. Golub and C. VanLoan, *Matrix Computations*, Baltimore, Maryland: Johns Hopkins University Press, 1983.